# REINFORCEMENT LEARNING FOR COLLISION AVOIDANCE

An Undergraduate Research Scholars Thesis

by

PAUL CROUTHER

Submitted to the Undergraduate Research Scholars program
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Research Advisor: Dr. Peng Li

May 2016

Major: Electrical Engineering

# TABLE OF CONTENTS

Page

# ABSTRACT

Reinforcement Learning for Collision Avoidance

Paul Crouther
Department of Electrical and Computer Engineering
Texas A&M University

Research Advisor: Dr. Peng Li
Department of Electrical and Computer Engineering

Autonomous travel poses challenges in machine learning navigation. Different approaches have been considered, such as reinforcement learning, dynamic programming methodologies, and other artificial intelligence assisted solutions. Dynamic programming methodologies exist to assist an agent, or robot, to interact with the environment based on predetermined environments. Real-world terrain and real-time complexity make learning and interacting with an environment particularly difficult. Reinforcement learning is a methodology that provides the agent with a learning system that is more environment and terrain independent. To this end, reinforcement learning is an effective way to solve the autonomous vehicle problem by proposing a model-free, or nearly model-free approach. The reinforcement learning algorithm used for this autonomous vehicle is Q-learning, which is an unsupervised learning algorithm where the agent explores from state to state until his goal is reached. With this algorithm, learning is achieved by sequential states of winning and losing, subject to obtaining the goal or failing to obtain the goal. The objective of this approach is to use particular goals to obtain navigation independent of the environment.

# DEDICATION

I dedicate this research to my wife, Elizabeth, and to the rest of my family. Without their continued support this opportunity would not have been possible.

# ACKNOWLEDGMENTS

I would like to thank Myung Seok, Seungjai, and Dr. Peng Li for their tremendous support. Their guidance has been paramount to the production of this thesis and my improvement as a student and person.

# CHAPTER I

# INTRODUCTION

## Motivation

Neurologically, behavioral studies have shown that animal behavior can be reinforced. Animals, when faced with a stimulus, show a strengthened response if their behavior is reinforced. The behavior that is reinforced is thus more likely to occur again the future. [1] Research suggests that reinforcement learning is the study of how the behavior of systems is optimized with respect to reward and punishments. To realize the system, a dynamic programming approach is utilized to obtain optimal outcomes. The reinforcement learning approach is an objective-based platform, which the agent acknowledges based on its own position, state, and goal. The agent will continue to explore states until a proposed goal is reached. For the Q-learning algorithm, particular alpha and gamma parameters can be chosen to change how the agent functions. Alpha, the learning rate, when close to 1, makes the agent only consider the newest information. Gamma, the discount factor, determines the shortsightedness of the agent such that immediate or future reward is considered. To learn, the agent treats an episode like a training session such that it explores the environment, represented by the Q-matrix, to find the most reward, until goal state is reached. For the agent to find the reward, the Q-matrix sets the current state to the initial state and then determines what action will produce the maximum Q value for the next state, and continues to set the current state to the next state until the goal state is achieved. In this thesis, we discuss the theory and implementations of a real-world system following these paramaters. We created such a system with an agent physical implementation on the Arduino and Parallax Shield platform, using a Raspberry Pi as the brain.

## Background

*An Introduction and Brief History*

To understand Q-learning in relation to the reinforcement learning problem, we have to evaluate the history of dynamic programming. The history begins with dynamic programming and Bellman's Principle of Optimality and the Bellman equation. Bellman's equation leads into Markov Decision

Processes as a method for solving stochastic dynamic programming problems with value iteration, which then provides a basis for Q-value iteration in reinforcement learning.

*Dynamic programming*

Initially, dynamic programming was introduced by Richard E. Bellman in 1940s as a process for dynamic decision making [I.1], but by 1953, it evolved into an recursively formulated mathematical optimization problem known as the Bellman equation [I.2]. To solve dynamic programming problems, Bellman suggested the Principle of Optimality. [2] The Principle of Optimality is such that if a dynamic programming problem can be broken down recursively, has at least an optimal substructure. The substructure is said to be optimal if for a given policy, all remaining decisions made must be with respect to the first initial decision, recursively. [2]

For a dynamic programming decision problem, we define a state at time $t$ to be $x_t$ with an initial $x_0$ state. We define an action $a_t$ in a set of actions at any time $a_t \in \Gamma(x_t)$. We also define a transition to a new state $T(x_t, a_t)$ when $a_t$ is chosen, and a payoff $F(x_t, a_t)$ with discount factor $\beta \in [0, 1]$. An infinite-horizon decision problem has the form:

$$V(x_0) = \max_{\{a_t\}_{t=0}^{\infty}} \sum_{t=0}^{\infty} \beta^t F(x_t, a_t) \tag{I.1}$$

subject to:

$$a_t \in \Gamma(x_t),\ x_{t+1} = T(x_t, a_t),\ \forall t = 0, 1, 2, \ldots$$

$$V(x_0) = \max_{a_0} \{F(x_0, a_0) + \beta V(x_1)\}$$

subject to:

$$a_0 \in \Gamma(x_0),\ x_1 = T(x_0, a_0).$$

$$V(x) = \max_{a \in \Gamma(x)} \{F(x, a) + \beta V(T(x, a))\}. \tag{I.2}$$

In modern times, dynamic programming is a method for solving problems with optimal substructures and overlapping subproblems. The problem can be solved in a recursive manner by breaking it into subproblems, and combining solutions to the overlapping subproblems. [3]

*Overlapping subproblems*

Subproblems that are related, and can be solved by using tabulation, are overlapping subproblems. The solution to each subproblem, may be encountered many times in the solving of the larger problem, in a bottom-up approach. [3]

*Optimal substructure*

A problem can be said to have optimal substructure if an optimal solution can be constructed efficiently from optimal solutions of its subproblems. [3]

*Markov Decision Processes*

The Markov Decision Process (MDP) was created by Ronald A. Howard, as an approach to solving sequential decision problems. The Markov Decision Process [I.3] contains a five-tuple $(S, A, R_a, P_a, \gamma)$, which consists of a set of possible states $S$, a set of possible actions $A$, a reward function $R_a(s, s') \to \mathbb{R}$, and a transition probability function $P_a(s, s') = Pr(s_t + 1 = s' | s_t = s, a_t = a)$, and a discount factor $\gamma \in [0, 1]$. The goal is to choose a policy $a_t = \pi(s_t)$ that maximizes the reward, weighted by the discounting factor. [4]

$$\sum_{t=0}^{\infty} \gamma^{\,t} R_{a_t}(s_t, s_{t+1}) \tag{I.3}$$

*Markov property*

The Markov Property is such that the effects of an action take in a given state only depend on that particular state, and not on prior history. This property will be used to take advantage of a value iteration in finding an optimal policy.

*Optimal policy and action selection*

To solve the MDP, a policy must be defined to choose a particular action, given a particular state. A policy, $\pi$, is a mapping from the current state to an action. Action selection can be either deterministic or stochastic. Within a policy, we select an action in our current state. For a

deterministic action, $T: \ S \times A \rightarrow S$ for each action and state, a new state is specified. The stochastic action is such that for each $T: \ S \times A \rightarrow \Pr(S)$.

*Choosing an optimal policy maximizing reward*

For an algorithmic implementation, we should exploit the MDP to generate a way to maximize expected reward. In general, the Markov Property of the MDP states that - the reward $R$ at time $t+1$, $R_{t+1}$ is dependent only on a state $S$ at time $t$, To find an optimal policy, we use value-iteration such as in equation I.5. Once we find an optimal policy, we can say we have exercised optimal control on an MDP. [5]

*Return and rewards*

For an MDP, a return is a sum of rewards, which can be modeled and evaluated in several ways. The finite-horizon model requires a terminal state which is always reached after some time T, on any given policy. The infinite-horizon model can be evaluated as either sum of discounted rewards, or the average reward of an unending MDP. The discount factor, $\gamma$, is applied to a sum of rewards, and must be $\gamma < 1$. [6]

*Finite-horizon reward model*

The finite-horizon reward model [I] is where we are only concerned with decisions and rewards up to time $t = H$. Assuming a finite action space, $A$, and a finite state space, $S$, the value of any policy $\pi$ is:

$$V_\pi(S_0) = E[\sum_{t=0}^{H} \gamma^{\,t} R(S_t) | \pi; S_0]$$

and we are interested in finding

$$\max_{\pi} \ \ V_\pi(S_0)$$
$$\pi = \{\mu_0, \mu_1, \ldots, \mu_{H-1}\}, \text{ where } \mu_i : S \rightarrow A$$

Given that for each $\mu_i$ there are $A^S$ possible mappings and for $\pi$ there are $(A^S)^H$ possible policies, we need to find a way find the value of all possible options. Recall the Markov property, where the past and the future are independent at a given a state, $S$. Recall that the Bellman equation can be solved for the policy function by performing backwards induction. Define the value function [I.4] at the $k$th time-step as:

$$V_k(s_k) = \max_{\mu_k, \mu_{k+1}, \ldots, \mu_{H-1}} E[\sum_{t=k}^{H} \gamma^{t-k} R(s_t)|s_k] \tag{I.4}$$

Similarly, we can find an optimal policy by working backwards from H.

$$V_k(s_k) = \max_{a \in A}[R(s_k) + \gamma \sum_{s'} P(s'|s,a) V_{k+1}(s')], \text{ where } V_H \equiv 0. \tag{I.5}$$

Following this theorem, it can be determined that value iteration gives a polynomial-time algorithm for the optimal policy, much reduced in dimensionality from $O(A^S)^H$ to $O(S^2(A)H)$ [5]

*Infinite-horizon reward model*

To find an optimal policy with an infinite-horizon [I.6], $H = \infty$, such that the value of any policy $\pi$ is:

$$V_\pi(S_0) = E[\sum_{t=0}^{\infty} \gamma^{\,t} R(S_t)|\pi; S_0] \tag{I.6}$$

If we want to prove suboptimality, we can use the Bellman back-up operator to bound the suboptimality of any incomplete run of value iteration as a function of backsteps used. [7] [5]

**Reinforcement learning**

Reinforcement learning is such that when the model is unknown, we consider new methods of finding optimal policies. In this case, there are three methods we will define. These algorithms are on-line and off-line algorithms. Off-line algorithms mean that the the actions that are performed are not controlled by the algorithm, or that the policy that performs actions is not controlled by the algorithm. On-line algorithms, like SARSA keep both the current and next state, and do

control the actions by policy. The three methods to be defined are temporal-difference (TD), state-action-reward-state-action (SARSA), and finally, Q-learning. [8]

*Temporal Difference*

Temporal-difference (TD) learning is a combination of Monte Carlo and dynamic programming ideas, by updating estimates based on other learned estimates. This technique is called bootstrapping. We focus on policy evaluation for the value function, given a policy. In finding an optimal policy, dynamic programming, Monte Carlo, and TD I.7 all use policy iteration. [8] [9]

In general, TD methods do not require a model of the environment, meaning it is a model-free algorithm. Consider policy evaluation:

$$V_\pi(s) = E[\sum_{t=0}^{\infty} \gamma^t R(s_t)|s_0 = s, \pi] \tag{I.7}$$

In our standard dynamic programming approach, we compute $V_\pi(s)$ by iterating:

$$\forall s: \quad V(s) \leftarrow R(s) + \gamma \sum_{s'} P(s'|s,a)V(s')$$

Now we consider a stochastic version:

$$\text{pick some state } s: \quad \text{sample} \quad s' \sim P(s'|s, \pi(s))$$
$$V_\pi(s) \leftarrow \alpha(R(s) + \gamma V'_\pi(s)) + (1-\alpha)V_\pi(s)$$

where $\alpha$ is step size. A possible step size: $\alpha_k = \frac{1}{k}$ for the $k$'th time we perform an update.

We can assure convergence of stochastic policy evaluation under the assumptions that every state visited infinitely often, and that $\alpha$ satisfies $\sum_{k=0}^{\infty} \alpha_k = \infty$; $\sum_{k=0}^{\infty} \alpha_k^2 < \infty$. We can prove this using the supermartingale convergence theorem.

In practice, a reason to use TD for policy evaluation could be that we do not have the transition model available. The samples are then generated by executing the policy, and performing the stochastic value function updates according to the states being visited.

TD only considers policy evaluation.[I.7] If we are interested in finding a (near-)optimal policy, we can use TD as the policy evaluation step in policy iteration:

Iterate:

1. Run TD to perform policy evaluation, which gives us $V_\pi(s)$, $\forall s$.
2. Pick a new policy $\pi$ as follows: $\pi(s) = \arg\max_a [R(s) + \gamma \sum_{s'} P(s'|s,a)V(s')]$

*Q-learning*

The Q-learning algorithm is an approximate version of value iteration where the expected value is approximated by sampling and simulation. [7] Consider the value iteration method with infinite-horizon rewards. [9] Recall the $Q$ function I.8:

$$Q(s,a) = R(s) + \gamma \sum_{s'} P(s'|s,a)V(s') \tag{I.8}$$

We can run dynamic programming (value iteration) by performing the Bellman back-ups in terms of the $Q$ function as follows:

Iterate:

$$\forall s,a : Q(s,a) \leftarrow R(s) + \gamma \textstyle\sum_{s'} P(s'|s,a) \max_{a'} Q(s',a')$$

Similar to the case of TD learning, we can run a stochastic version instead:

For $k = 0, 1, 2, \ldots$

Sample some $s,a : Q(s,a) \leftarrow (1-\alpha_k)Q(s,a) + \alpha_k(R(s) + \gamma \sum_{s'} \max_{a'} Q(s',a'))$

Initialize $Q(s,a)$ arbitrary ($\forall s,a$).

1. For $t = 0, 1, 2, \ldots$

Choose the action $a_t$ for the current state $s_t$. (E.g., by using an $\epsilon$ greedy policy.)

Take action $a_t$, observe $R(s_t)$, $s_{t+1}$

$$Q(s_t, a_t) \leftarrow (1 - \alpha_t)Q(s_t, a_t) + \alpha_t[R(s) + \gamma \max_a Q(s_{t+1}, a)] \qquad (*)$$

[5]

*State-Action-Reward-State-Action*

Sarsa [I.9] is almost identical to Q-learning. One of the differences between Sarsa and Q-learning is in the Q-function update: I.8 becomes:

$$Q(s_t, a_t) \leftarrow (1 - \alpha_k)Q(s_t, a_t) + \alpha_k[R(s) + \gamma Q(s_{t+1}, a_{t+1})] \tag{I.9}$$

Here $a_{t+1}$ is the action the agent ends up taking (which is not necessarily $\arg\max_a Q(s_{t+1}, a)$, as the policy includes randomness). [5]

*Eligibility traces*

An eligibility trace is a temporary record of the occurrence of an event, taking an action, or even visiting a particular state. The trace makes a record of the parameters regarding a particular event as eligible for learning changes. In general, the eligibility trace is a great mechanism for reinforcement learning by combining it with any method to improve learning. There are two different views we adopt to understand eligibility traces. [8] [10]

1. The forward view is where we consider a theoretical view of what particular methods compute. We look forward into the future to decide how to update each state based on future reward and states.

2. The backward view is more focused on the credit assignment, or a mechanistic understanding of the algorithm itself. The traces record which states have been recently visited, based on $\gamma\lambda$ decay.

*Tile coding*

Tile coding is a partitioning of elements called tiles into receptive fields for some binary feature. We can use tile coding to directly represent the number of tilings as the number of features. This method holds a significant advantage over approximate value function in gradient-descent function approximation. [11]

*Summary*

For our purposes, we focus on the Q-learning algorithm implementation. We choose this algorithm primarily because:

1. Q-learning is an off-policy method for finding optimal policies like value iteration.
2. If there is no explicit model of a system this method can be used directly.
3. Instead of approximating the cost function of a particular policy, it updates Q-factors associated with an optimal policy which avoids multiple policy evaluation steps. [7]

In our approach, we also adopt tile coding and eligibility traces to aid in Q-value convergence.

# CHAPTER II

# ALGORITHMS

## The reinforcement learning problem

The reinforcement learning problem is phrased as a way to learn from interactions in order to achieve a goal. In particular, we define a learner and decision-maker as the agent. This agent interacts with the environment, which is encompassed by everything outside of the agent. [8] Specifically, the agent and environment interact by learning with trials of exploration and exploitation. This is discussed in more detail in chapter IV.

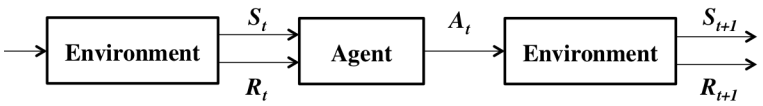


Fig. II.1.: Figure of agent-environment interaction.

## Using Q-learning to solve the reinforcement learning problem

In the previous chapter, we used the history of reinforcement learning to inform our understanding of Q-learning. We selected Q-learning as the algorithm methodology of choice because of summary at the end of the chapter I. This chapter focuses on the algorithmic implementation of the Q-learning methodology.

### *Q-learning state-action space*

In general, we can take advantage a Q-learning algorithm [Algorithm 1] can be simplified to a set of actions for any given state. For example, given a position of Cartesian coordinates for an agent in $(x, y) \in R^2$, a set of possible actions in the direction angle $\theta$. In general, for a simple implementation, the state and action values are initialized to zero. Then, the agent searches the state and action space until an outcome is determined. The Q-value, is then increased if the action of the outcome is better for the given state. Consequently, if the action is not better for a given state, the Q-value is reduced. Q is only updated after the outcome is seen.

*Q-table*

The Q-table is the look up table consisting of all parameters of a given state and action. Each Q-table entry consists of a set of actions. We use parameters epsilon, alpha, gamma, and iteration number to help define how the agent interacts with the environment.

**Parameters**

A more holistic list of parameters, such as $\alpha$, $\gamma$, and $\epsilon$ are listed in table III.1, and their effects discussed in chapter IV.

*Epsilon*

The $\epsilon$ parameter is used to manipulate agent exploration by determining the randomness of action selection in the agent. It works by choosing the next action in a stochastic manner if it is greater than $\epsilon$. Consequently, this feature can be problematic when the epsilon value is too high after all options have been explored. When $\epsilon = 0$, the agent enters exploitation phase because it only chooses from the Q-table. [6]

*Alpha learning rate*

The alpha parameter determines the learning rate of the agent. If $\alpha = 0$, then the agent will learn nothing, where $\alpha = 1$, the agent will only consider the most recent information. It is generally optimal if $\alpha = 1$ when the problem is deterministic, but with a stochastic problem, the learning rate is generally set to some constant, such as $\alpha = 0.1$. [6]

*Gamma discount factor*

The discount factor known as $\gamma$, determines the importance of future rewards. The agent with a $\gamma = 0$ will be more "greedy" or short-sighted, only considering the current rewards. With $\gamma = 1$, the agent considers the highest long-term reward. [6]

**Algorithm 1: Function Q-Learning** $(S, A, R, S', Q)$

**Input**: $S$ is the last state, $A$ is the last action,$R$ is the immediate reward received, $S'$ is the next state, $Q$ is the array storing the current action-value function estimate.

1. $\delta \leftarrow R + \gamma * max_{a' \in A} Q[S', a'] - Q[S, A]$
2. $Q[S, A] \leftarrow Q[S, A] + \alpha * \delta$

**Output**: $Q$

*Summary*

In summary of this chapter, we discussed the reinforcement learning problem, a Q-learning solution and theoretical implementation. In the following chapter, we will build a software agent-environment interface that implements the theory and groundwork from this chapter.

# CHAPTER III

# SETUP

## Learning environment

In the previous chapter II, we defined an agent-environment interface. In this chapter, we focus on the creation of this environment in software by using open source libraries and programming environments. In addition, for a real-world implementation, there are hardware constraints that must be considered. This chapter lays the framework for the software and system platform.

### *Creating an environment*

The environment for reinforcement learning requires a value system of punishments and rewards. Punishments are defined as negative reward, which opposes the agent's mission to generate the highest possible reward. Rewards are used to control the goal state and action selection of the agent. We create an environment conducive to reinforcement learning by providing objects and actions of reward and punishment. [8]

## Learning agent

### *Creating an agent*

An agent should learn in a given environment using reinforcement learning. For our purposes, the agent is controlled with a two-wheel setup for simplicity, providing a platform for a set of actions given an agent's state in the environment. This precedent holds true for both the simulation and real-world system. An agent should have the freedom to move, with respect the environment, providing a mode of learning for an episode and test. We create an agent from the simple two-wheel arrangement, which provides a forward-backward movement to each motor, driving the wheels. Controlling the action taken is as simple as sending a signal to one or both motors. The simulation aspect of the agent in an environment is shown in figure III.2

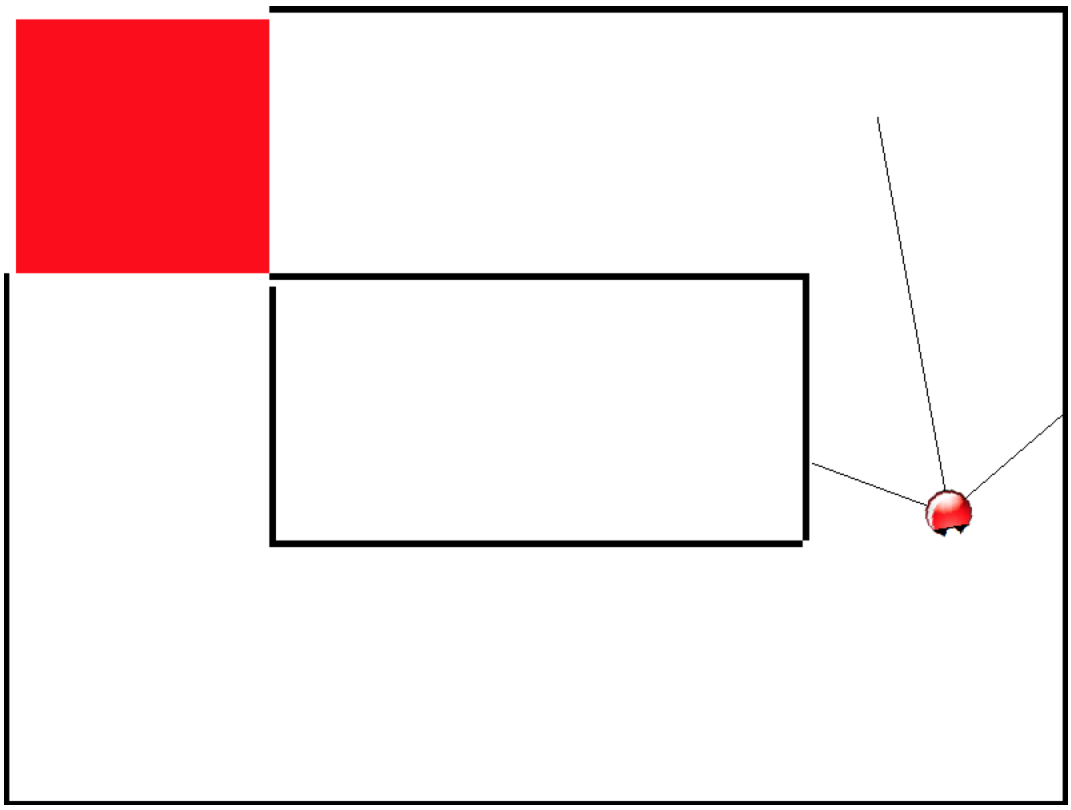

Fig. III.1.: An agent interacting with the learning environment.

*Reinforced behavior*

As discussed in chapter II, we defined the interaction between the agent and environment. For our environment, punishment corresponds to hitting an object, and movement. A successful arrival generates a reward for the episode. The process generates a Q-table of state-actions with given reward for the agent based on the environment and actions taken given a state. The agent is encouraged to explore more of the environment based on greed. Specifically, we can tune the agent's learning behavior based on epsilon, alpha, and gamma, the discount factor. This is discussed in chapter IV and chapter V.

**Sensor embedding**

How the agent spatially understands its environment requires embedding spatial parameters for action-state selection. For instance, the location, direction, and speed are important parameters for state and action. Given a state and action, the agent learns based on using its own location, speed,

and direction. In particular, the state and action pairs use relative coordinates for location, speed, and angle for direction.

*Movement*

We define three possible movement states to add another distinguishing feature for state and action selection. For our purposes, right, left, and forward are used as discrete movement states for the agent based on return of reward. Challenges faced with movement embedding are servo calibration and servo speed. With the servos supplied in the Parallax Shield kit, we give the agent directional commands by manipulating the voltages supplied to each left and right servo. To this end, we obtained full forward movement, slight right turn, and slight left turn.

*Location*

Location embedding is difficult due to challenge of limited physical knowledge of the agent, of itself with respect to obstacles. This challenge is particularly prevalent in the first iteration and early stages of learning. Other methods commonly employed include initial value estimates and optimistic initial values. [8] For our purposes, our conditions are described in table III.1.

The Cartesian coordinates of a grid in the testing environment determine the location coordinates in the simulation. Discretization is performed on the Cartesian values to pixel-level precision, which the agent uses for action selection. We populate the Q-table with an accrued reward, indexed by exploration iteration, or episodes.

In general, location also poses a real-world challenge in complexity, so we avoid continuous interpretations of location.

*Direction and angle*

Direction and angle, while easy to implement in simulation, is a real-world challenge. Specifics about angle generate problems of complexity, and continuous interpretations should be avoided.

We also perform a discretization of the angle to a set of directions determined by quadrants. This is more reasonably implemented in hardware, and simplifies the state and action selection.

*Speed*

The last spatial parameter of the agent in a given environment, is speed. The speed of the agent may also be accounted for a given state and action, but continuous interpretations should be avoided due to complexity. The speed entry is given by a relative measurement of the servo speed in a given direction. Discrepancies between the agent in hardware and simulation pose difficulty with knowledge of the real speed of the agent and the estimated speed. Estimation of wheel rotation and speed propagates error while increasing the size of the Q-table.

**Sensing hardware**

Sensing gives the agent feedback about the environment, which is crucial to learning outcomes. There are a few sensing mechanisms we have implemented on the agent.

*Wire antennae*

For collision detection at a given boundary, we use Parallax "whiskers" [12] that produce a high signal when touched, causing the agent to be punished. The wire antennae also protects the agent from excessive frontal shock while also providing collision detection.

Initially, force sensors were mounted as the collision detection scheme for the agent. We depreciated this idea due to physical infeasibility and sensor threshold impracticalities with continuous readings. The simplicity and robustness of the whiskers were more appealing, which subjects the agent to collision detection state parity.

*Line sensor*

In addition, we used a simple line sensor [13] to also detect target arrival, and to send the agent back to the starting location. We constructed a tape periphery around the environment as well as the the target location.

*Ultrasonic sensors*

To collect distance measurements with boundaries, we opted for the ultrasonic distance sensor [14] scheme, which allows the agent to detect distance based on ultrasonic pulses. The schematic for the ultrasonic sensor is intricate because the pulses should to be sent at different times to avoid confusion. On the agent, the sensors are mounted in a direct forward, left, and right position, with the last two sensors placed at 45 degrees between the forward-left and forward-right sensors. Using a sequential pulsing scheme, the ultrasonic sensor sends a pulse at different times in different directions, receiving information about the agent's relevant location. If the ultrasonic signal interpolates, then location is difficult to decipher. The sensor configuration is shown in figure III.2.

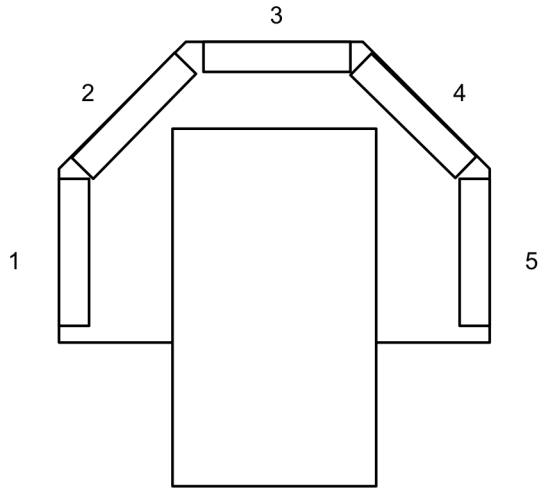


Fig. III.2.: Ultrasonic sensor configuration.

**Robot architecture**

We used a matrix of components for the hardware aspect of the robot architecture, shown in figure III.4. The components provided the basis from which we performed real-world testing. We based the robot on a shield kit for the Arduino, called the Parallax Shield kit.

*Using a shield*

An Arduino shield [15] is a component that attaches directly to the general-purpose-input-output, or GPIO pins and other connections such as battery and power. With a shield connected, the Arduino

becomes a part of the robot that can be easily interfaced with other components, in particular, without using more GPIO pins than necessary. Upon the shield, we build the rest of the robot agent.

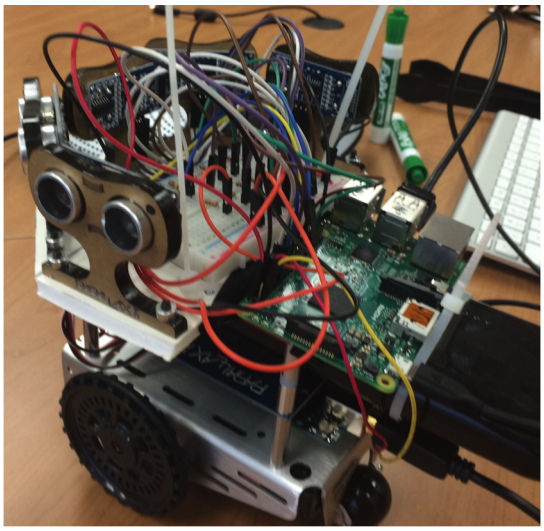

Fig. III.3.: Picture of the assembled robot hardware, used for testing.

*Raspberry Pi 2*

The brain of the agent is the Raspberry Pi 2 [16], which has a 900 MHz quad-core ARM Cortex-A7, 1GB DDR3 RAM, 4 USB ports, and 40 GPIO pins. [17] Each ultrasonic sensor has VDD, GND, and two supporting pins for sensing. The GPIO pins support all five ultrasonic sensors. It is powered by an external battery. The code used for the Raspberry Pi is included in the appendix.

*Arduino*

The Arduino Uno [18] serves as the controller for some of the sensing hardware. The Arduino Uno runs on a ATmega328P Microcontroller, operating at 16 MHz. It has 20 GPIO pins, and has 32 KB of flash memory. Originally, the Arduino controlled the ultrasonic sensors, as well. We moved the ultrasonic sensors, and removed the serial port connection due to issues described in the Raspberry Pi 2 and Arduino serial connection section, The Arduino is used to send and receive I2C messages, as well as control the robot's servos via Parallax Shield. [19] Code used for this setup is displayed in the appendix.

*Parallax Shield*

The Parallax Shield [15] for the Arduino is a relatively inexpensive platform that gives reasonable access to a large set of libraries and examples. In addition, the Parallax provides the user with a lot of customization options. With the Parallax Shield kit, we have a well-supported platform for the robot agent, and support for future work.

*Servos*

The servos serve a non-traditional purpose, as motors for control and movement. The servos are supplied with the Parallax Shield kit, and have continuous motion and rotation. They can be rotated and controlled using the Arduino IDE interface with movement functions called from an included Servo.h library.

*Raspberry Pi 2 and Arduino Serial Connection*

For the hardware aspect of the reinforcement learning platform, the Raspberry Pi is used to log and control simulation and iteration, and an Arduino is used for motor and sensory control. The Raspberry Pi is realized as the controller for the Arduino. Initially, information was passed between the Raspberry Pi 2 and the Arduino via serial port interaction.

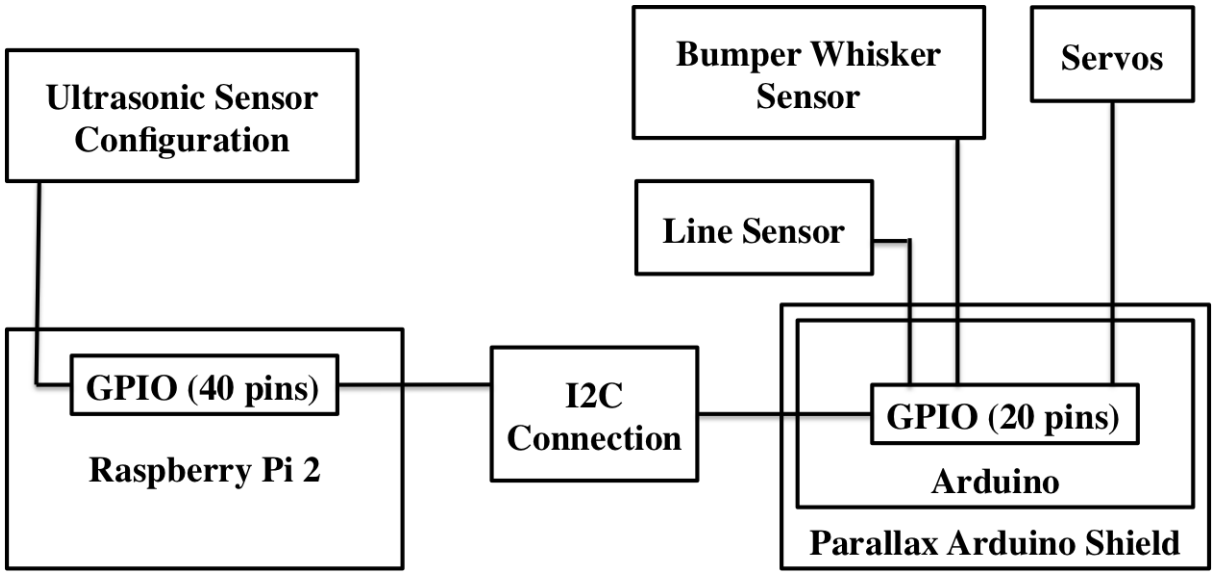


Fig. III.4.: A general high-level view of the robot hardware architecture.

Ultimately, we adopted an I2C connection over the serial USB port connection. Corruption on the serial port made it difficult to send and receive strings, as there were a few characters missing from the message. In addition, we could not send and receive multiple messages simultaneously, providing a bottleneck for data traffic between the Arduino and the Raspberry Pi.

*External power*

We use two external battery sources to power the servos, Raspberry Pi, Arduino, ultrasonic sensors, and other components. For the Raspberry Pi, which is essentially the brain, we use an Ankar Astro E5 battery which supplies 16000 mAh and USB functionality. For other components connected to the Arduino, we the use the 4-cell battery pack supplied with the Parallax Shield kit. The 4-cell battery pack powers the Arduino and the shield, which powers the servos. With the Astro E5 battery, we receive power for about 10 hours of testing.

**Software environment**

*A gaming environment for simulation*

We use the Python wrapper, Pygame [20], to implement the simulation of the agent in a given environment. The Pygame library gives access to modules that construct the game activity. The game activity is the instantiation of the learning environment described at the beginning of the chapter III.

*Modules*

The process begins with the import the necessary modules, such as Pygame, in an effort to create the software implementation of a reinforcement learning. To this end, it is important to note Pygame requires a screen initialization, so we initialize the screen and set the screen size. We initialize by calling the initialization method on the Pygame instance, then setting the display size. Screen size and granularity is highly relevant to the discussion of discretization versus continuous domains.

Table III.1: Members of class Agent

| Parameter | Description | Initial value |
|---|---|---|
| Epsilon $\epsilon$ | Stochastic factor | 0.05 - 0.15 |
| Alpha $\alpha$ | Learning rate | 0.025 - 0.15 |
| Gamma $\gamma$ | Discount factor | 0.7 - 0.9 |
| Rewards | Initialized reward | 0 |
| Count | The exploration phase counter | 0 |
| Iteration | Max episodes before ending exploration | 3000 |
| Count test | The testing phase counter | 0 |
| Iteration test | Max iterations in the test phase | 100 |
| Number arrivals | Total successes in exploration | 0 |
| Number test arrivals | Total successes in testing | 0 |

*Initialization and display size*

The display size is important because it is the boundary of the simulation. The size of the screen, and the amount of pixels used, has some correlation with agent performance. The size of the screen, which is effectively the size of the simulation environment, determines exploration complexity. Due to discretization, the complexity of an environment corresponds with the discretization schemes and granularity. For the agent, tile coding provides necessary discretization for action selection, but the performance is still impacted.

*Python implementation*

We optimized and redeveloped open source code for the software implementation of Q-learning in python. This basis enables us to create a car object, with the Q-learning parameters of $\epsilon, \gamma, \alpha$, states, actions, and iterations. This object instantiation is discussed in table III.2.

*Choosing an action*

We generate state-action pairs into a look-up table, where, the dimensionality of the table is determined by the number of different actions and states. The look-up table can have a size of $S_t x A_t$, also described in the finite-horizon reward model in chapter I. To choose an action in

Table III.2: Methods of class Qlearning

| Method | Description |
|---|---|
| Decreasing parameter | Decreases $\alpha$ and $\epsilon$ based on iterations |
| Get Q | Returns a Q-value |
| Learn Q | Learns the new Q-value |
| Choose Action | Determines action selection in a given state |
| Learn | Gets max reward and updates with Learn Q |

exploration, the agent can decide based on the state-action values stored in the look-up table. A caveat is that we also need to consider actions with the same value. In order to choose, we pick randomly between actions of the same value. We use epsilon to choose by checking if random value is less than epsilon, incorporate those values into the Q-value for that particular state. [21] Code available in the appendix section.

*Qlearning object and methods for state-action selection*

In our software simulation, we used the class Qlearning and called a constructor to generate an object of type Qlearning named carAgent. The object has methods that enable reasonable access to state and action selection. The Qlearning class methods directly implement the Q-learning algorithm. [21] Code is also available in the appendix section. Methods are described in table III.2.

*Summary*

In the summary of this chapter, we define software and hardware tools that were used to create our platform. This software agent generates a car agent object that contains all of the state-action parameters of the Q-table, such as state, action, reward, and controlling parameters such as, alpha, gamma, and epsilon. In hardware, the challenge is to create a hardware platform that supports the software environment. We defined a reasonable sensor interface, that gives the hardware agent, or robot, knowledge about the real environment. This provided mapping challenges between the software interface and the continuous, real-world implementation. In the next chapter, we run simulations and collect learning data from the agent in the simulation environment.

# CHAPTER IV

# METHODS

## Collecting data

In the previous chapter, we produced the Q-learning algorithms and a hardware interface for testing. In this chapter, we will introduce the methods used for collecting and evaluating the data from the simulation. The data is evaluated in two phases, exploration and exploitation. This principle is best illustrated in slot machines, and the strategies used to achieve the greatest reward.

## N-armed bandit problem

We generate a game in which we consider a one-armed bandit, or slot machine. We adopt a strategy to obtain information about the slot machine game. For our purposes, the strategy adopted is $\epsilon$-greedy in exploration. Initially, there is no knowledge about the machine, so a player enters the exploration phase to obtain information. Likewise, when enough information has been obtained, the player exploits the known information to achieve the greatest award. [22] [23]

### *Exploration*

Exploration is when the agent, given some epsilon, $\epsilon$, continues to explore the environment in order to gain more information about expected payoff. In the slot machine analogy, we attempt to select the best lever for the slot machine with 1-$\epsilon$ of the trials, and select at random a proportion of $\epsilon$ trials with uniform probability.

### *Exploitation*

Exploitation is when the agent aims to receive the highest expected payoff. This differs from exploration, where the agent is still looking for additional information. In exploitation, we set $\epsilon$ to 0.

**Exploration and exploitation trade-off**

In the bandit problem, there is difficulty with decision making under uncertainty. The decision maker must choose what either seems to be the best choice, which is exploitation, or test and explore for some alternative which could be better than the current best choice. [24]

*How and what we choose*

Using Q-learning, we generate a Q-table from the exploration phase. Then based on epsilon greedy, the agent can choose either an optimal action or random action in the testing phase.

*Alpha learning factor sweep*

In figure IV.1, we swept $\alpha$ from 0.025 to 0.15, to see how it impacts the learning outcome of the agent. As expected from the simulation, the higher the learning rate, the more the agent strives for higher long-term reward. With a lower learning rate, the agent produces quicker, more "greedy" learning outcomes. We maintained that $\gamma$ is 0.9, $\epsilon$ is 0.1, with max episodes of 3000.

*Gamma and epsilon sweep*

In figures IV.3 and figure IV.2, we swept $\gamma$ from 0.7 to 0.9, and $\epsilon$ from 0.05 to 0.15, to see how it impacts the learning outcome of the agent. As expected, there was little effect in this manner on the success outcome. We can use the discount factor and epsilon to define a decreasing parameter.

*Exploitation: Testing phase results*

For our purposes, there are two kinds of test phase in which the agent exploits the Q-table.

1. In the simulation test phase, we run some number of episodes in test phase with the agent solely exploiting the Q-table from knowledge gained during exploration.

2. There is a test phase where we use intelligence gained to test in the real-world. To use the intelligence gained from the simulation in a real system, the Q-table is loaded into a real robot agent in the form of an object, carAgent, of type Qlearning, and we use the methods of its instantiation to exploit the Q-table by using sensing and moving hardware. In our case,

the sensor and movement hardware is controlled using the a Raspberry Pi 2 for the brain, and an Arduino for servo and motor control. Then, the knowledge gained during exploration is exploited during a test phase with these real values as opposed to hard coded software parameters.

In the simulation testing phase, we run 100 episodes of exploitation, where $\epsilon$ is 0, and the agent only exploits what it already learned about the environment. From the data collected, there was a 100% success rate during the majority of tests performed after 3000 learning episodes in exploration. In the hardware agent implementation, we call the carAgent methods to perform actions such as moving left and right by sending pulses to the servos, and detecting collisions with the whiskers. With this version of testing, we also saw convergence, but at a slower rate, due to the speed of the servos. There is more data to collect in this area.

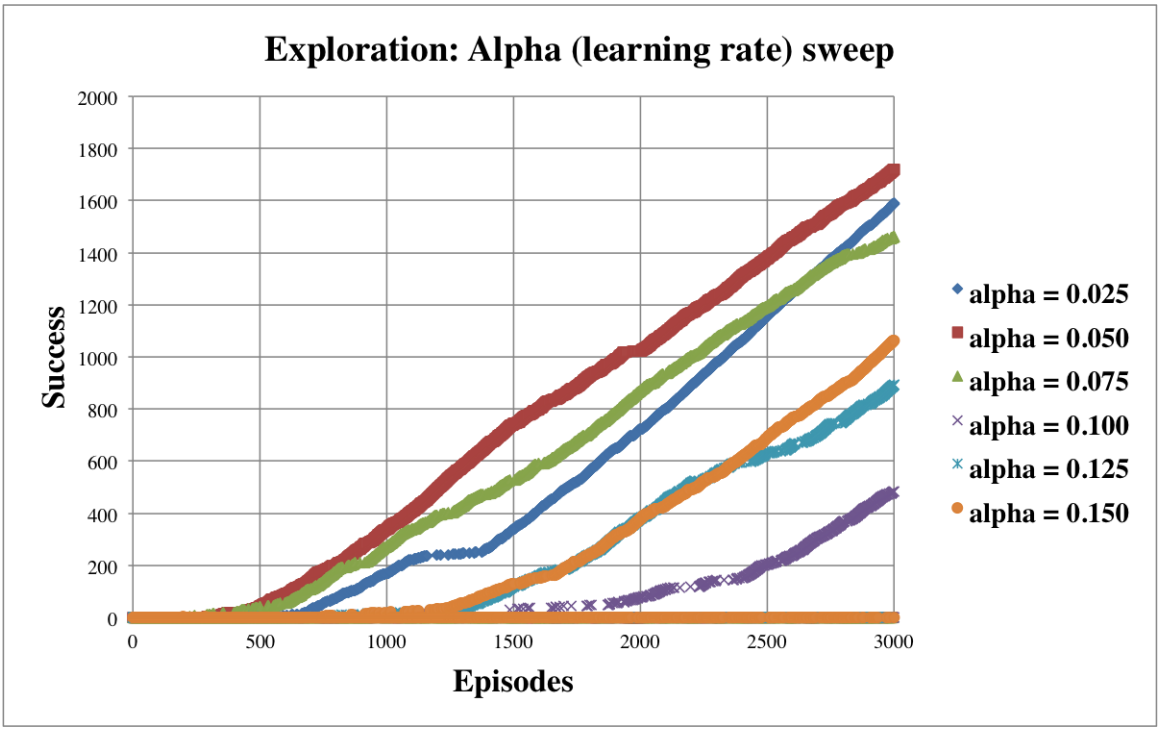


Fig. IV.1.: Exploration: Sweeping values of alpha from 0.025 to 0.15.

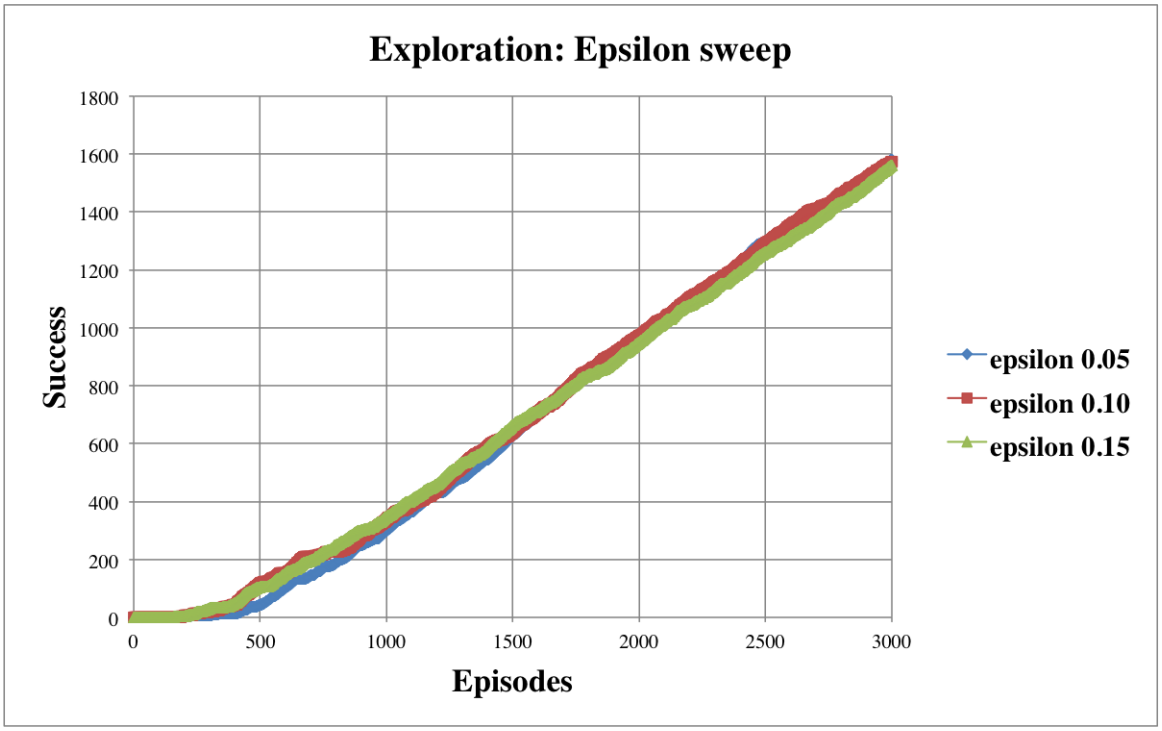


Fig. IV.2.: Exploration: Cumulative success with epsilon sweep from 0.05- 0.15.

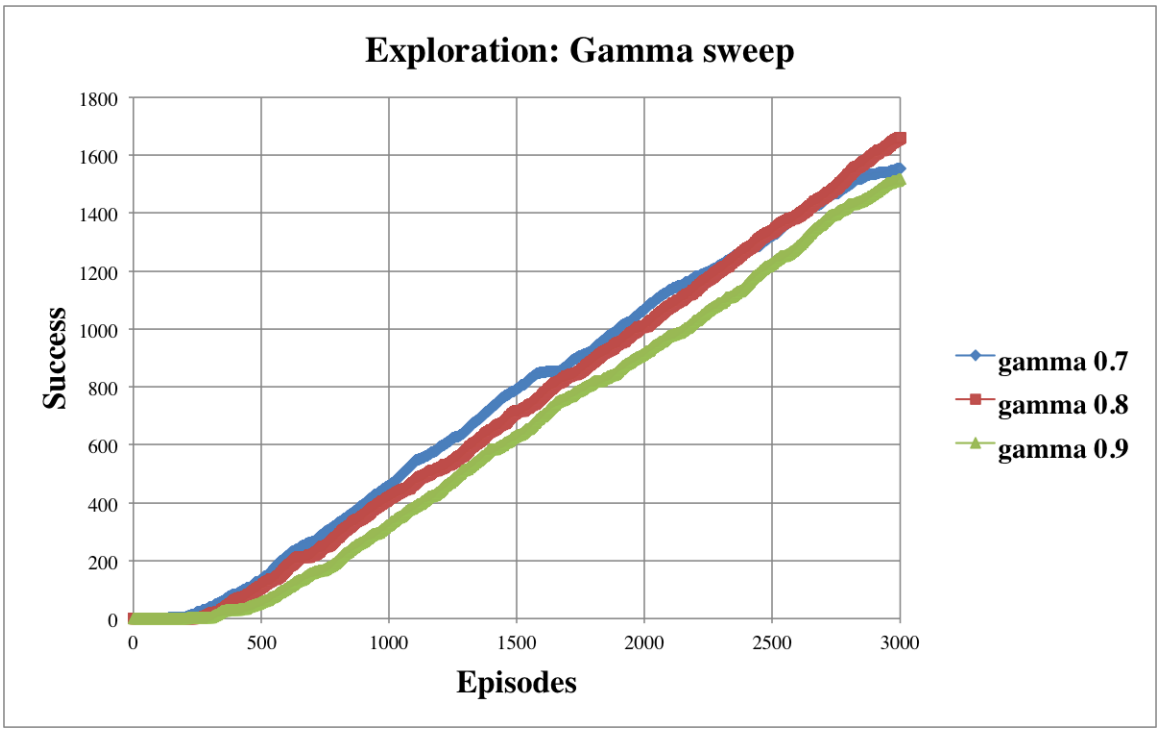


Fig. IV.3.: Exploration: Cumulative success with discount factor sweep from 0.7- 0.9.

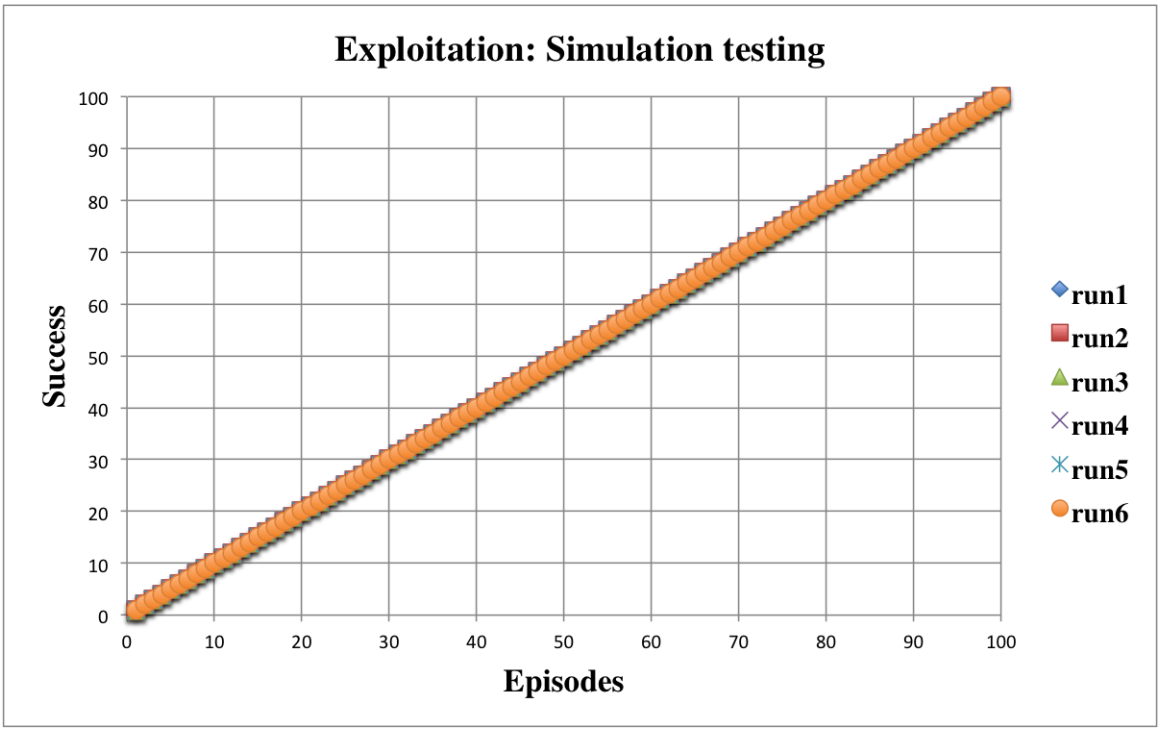


Fig. IV.4.: Exploitation: Testing phase in simulation with some results showing 100% success.

*Summary*

In this chapter, data was collected in the simulation and tested on the hardware agent and software agent platforms. In the software test phase, we generated a very high success rate in testing after learning episodes. In general, the simulation shows that it may be possible to lower iterations in order to reduce learning time, while maintaining a very high success rate in test. In the next chapter, with more detail, we discuss the conclusions of exploration and exploitation in software and hardware.

# CHAPTER V

# CONCLUSION

## Analysis

In this chapter, we discuss the results collected from the previous chapter, chapter IV. The results show that after the agent arrival success for reaching the target increases every episode. In some cases, it does not arrive at the target until it has crashed more than 300 times. With this in consideration, we can choose to look at the data collected after the agent has reached the target at least once. Initially, the calculation for the success rate is as follows:

$$\frac{Number\ of\ successes}{Total\ episodes}$$

We can conclude a higher arrival success rate this from performing a removal of the first few iterations of crashes before the agent arrives its first time. The new process is shown by the following formulation:

$$New\ success\ rate = \frac{Number\ of\ successes}{Total\ episodes\ -\ Episodes\ before\ first\ arrival}$$

Looking at data from the simulation, we see that a sweep of $\alpha$, the learning factor, corresponds with achieving higher success in the exploration phase. It also shows a slower convergence to the higher amount of success with changes in $\alpha$. This can be attributed to the large role that the learning factor plays in the "greed" or shortsightedness of the agent. With a more greedy agent, and an $\alpha$ closer to 0, the agent performs by trying to get the greatest award quickly. The agent is less concerned with the larger overall reward. In contrast, $\alpha$ closer to 1, the agent considers a longer term reward in exploration, meaning when it convergences more slowly, in the end leaves more reward. It can also be determined that $\alpha = 0.05$ is a very good potential learning rate for future work on this project.

The most ideal outcome results in a simulation test phase, where many parameters are known and controlled. In the real world, choosing good parameters is a challenge, and makes exploration

difficult. Other challenges include localization, time constraints, and changes to the environment. The next steps of this research involve using new technologies, previously unavailable, to solve these challenges.

*Summary*

In this chapter, we briefly discussed new ways to interpret the data that was gathered, as well as how parameters impacted the learning outcomes of the agent. In the final chapter, we discuss new features to be added and implemented to increase the overall system performance. As the work is ongoing, there will be future works based on this software and hardware robot platform.

# CHAPTER VI

# FUTURE WORK

## Future considerations

There are a few takeaways from the study and analysis of this research. They can be summarized as:

1. Analyzing the data from the simulation, we can determine that changing $\alpha$, or the learning rate significantly impacts the greed of the agent in exploration. We can also determine that the $\gamma$ and $\epsilon$ parameters show varying degrees of impact on the exploration.
2. Taking a software approach, it is reasonable to make an agent-environment model that can be explored and tested.
3. In general, it is reasonably difficult to implement the hardware aspect of the software platform.
4. From a hardware point of view, one must be careful when mapping the sensors to action selection within a given state. Using approximation and discretization methodologies, we can generate a software agent that closely correlates with hardware.

Using new approaches, such as clever discretization schemes and approximations, problems with localization and dimensionality are surmounted. For our purposes, because the robot was not very large, and the environment was also reasonable in size, we did simulation and testing indoors. This poses a problem with the agent knowing its own location. If the project is scaled up to train and test outside, the robot could take advantage of GPS technology. By giving a reasonably accurate indication of the agent location, the state-action selection in the real system is given an additional feature to assist with learning.

# REFERENCES


[1] C. B. Ferster and B. F. Skinner, "Schedules of reinforcement.," 1957.

[2] R. Bellman, *Dynamic Programming*. Princeton, NJ, USA: Princeton University Press, 1 ed., 1957.

[3] T. H. Cormen, *Introduction to algorithms*. MIT press, 2009.

[4] R. Howard, *Dynamic Programming and Markov Processes*. Technology Press Research Monographs, MIT Press, 1960.

[5] J. Schulman and P. Abbeel, "Berkeley cs294 lectures." https://inst.eecs.berkeley.edu/~cs294-40. Accessed: 2015-12-10.

[6] C. J. C. H. Watkins, *Learning from delayed rewards*. PhD thesis, King's College, Cambridge, 1989.

[7] D. P. Bertsekas, *Dynamic programming and optimal control*, vol. 1. Athena Scientific Belmont, MA, 1995.

[8] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*, vol. 28. MIT press, 1998.

[9] C. Szepesvári, "Algorithms for reinforcement learning," *Synthesis lectures on artificial intelligence and machine learning*, vol. 4, no. 1, pp. 1–103, 2010.

[10] S. P. Singh and R. S. Sutton, "Reinforcement learning with replacing eligibility traces," *Machine learning*, vol. 22, no. 1-3, pp. 123–158, 1996.

[11] A. A. Sherstov and P. Stone, "Function approximation via tile coding: Automating parameter choice," in *Abstraction, Reformulation and Approximation*, pp. 194–205, Springer, 2005.

[12] "Parallax whiskers." http://learn.parallax.com/node/235, 2012. Accessed: 2016-04-09.

[13] "Sparkfun redbot sensor-line follower." https://www.sparkfun.com/products/11769?gclid=CPjV4tzFwMgCFYkBaQodDw4Hxw/, 2016. Accessed: 2016-04-09.

[14] "Hc-sr04 ultrasonic module." http://forum.arduino.cc/index.php?topic=37712.0, 2016. Accessed: 2016-04-09.

[15] "Robotics shield kit(for arduino)." https://www.parallax.com/product/130-35000, 2016. Accessed: 2016-04-09.

[16] "Raspberry pi 2 model b - raspberry pi." https://www.raspberrypi.org/products/raspberry-pi-2-model-b, 2016. Accessed: 2016-04-09.

[17] M. Richardson and S. Wallace, *Getting Started with Raspberry Pi*. " O'Reilly Media, Inc.", 2012.

[18] "Arduino uno r3." https://www.arduino.cc/en/Main/ArduinoBoardUno, 2016. Accessed: 2016-04-09.

[19] M. Banzi and M. Shiloh, *Getting Started with Arduino: The Open Source Electronics Prototyping Platform*. Maker Media, Inc., 2014.

[20] P. S. et al., "Pygame." http://pygame.org/, 2011.

[21] T. Dewolf, E. Hunsberger, and T. C. Stewart, "Reinforcement learning open source code." https://github.com/studywolf/blog/tree/master/RL. Accessed: 2015-09-15.

[22] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine Learning*, vol. 47, no. 2, pp. 235–256.

[23] H. Robbins, "Some aspects of the sequential design of experiments," in *Herbert Robbins Selected Papers*, pp. 169–177, Springer, 1985.

[24] J.-Y. Audibert, R. Munos, and C. Szepesvári, "Exploration-exploitation tradeoff using variance estimates in multi-armed bandits," *Theoretical Computer Science*, vol. 410, no. 19, pp. 1876 – 1902, 2009. Algorithmic Learning Theory.

[25] J. Stober, *Sensorimotor Embedding: A Developmental Approach to Learning Geometry*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, Austin, TX, May 2015.

# APPENDIX A

# CODE

## Learning.py

```
import random

class QLearning:
    def __init__(self, actions, epsilon, alpha, gamma, iterations):
        self.q = {}
        self.epsilon = epsilon
        self.alpha = alpha
        self.gamma = gamma
        self.actions = actions
        self.iterations = iterations

    def deacreasing_parameter(self):
        if(self.alpha > 0):
            self.alpha = self.alpha - self.alpha/self.iterations
        else:
            self.alpha = 0
        if(self.epsilon > 0):
            self.epsilon = self.epsilon - self.epsilon/self.iterations
        else:
            self.epsilon = 0

    def getQ(self, state, action):
        return self.q.get((state, action), 0.0)
        #return self.q.get((state, action), 0.0)
        # return self.q.get((state, action), 1.0)

    def learnQ(self, state, action, reward, value):
        oldv = self.q.get((state, action), None)
        if oldv is None:
            self.q[(state, action)] = reward
        else:
            self.q[(state, action)] = oldv + self.alpha * (value - oldv)
            print (self.q[(state, action)])

    def chooseAction(self, state):
        if random.random() < self.epsilon:
            action = random.choice(self.actions)
        else:
            q = [self.getQ(state, a) for a in self.actions]
            maxQ = max(q)
            count = q.count(maxQ)
            if count > 1:
                best = [i for i in range(len(self.actions)) if q[i] == maxQ]
                i = random.choice(best)
            else:
                i = q.index(maxQ)

            action = self.actions[i]
        return action

    def learn(self, state1, action1, reward, state2):
        maxqnew = max([self.getQ(state2, a) for a in self.actions])
        self.learnQ(state1, action1, reward, reward + self.gamma*maxqnew)
```

# Q471.py - Part 1

```
#V4.71
#Real Model-free Reinforcement Learning
#Learning phase for 3000 iterations and test phase for 100 iterations
#ARRIVAL : +10, COLLISION : -10, move : -0.1
#+100,-10,-0.1 / 60%
#MORE DISCRETIZATION -> MORE STATES

import warnings
warnings.filterwarnings('ignore')
import learning, pygame, time, sys, Sprite
import math
import xlsxwriter
import json
import cPickle

sys.setrecursionlimit(15000)

pygame.init()

# DISPLAY CONSTANTS
fps             = 60
display_width   = 800
display_height  = 600
car_width       = 35
car_height      = 35
black           = (0,0,0)
white           = (255,255,255)
red             = (255,0,0)
blue            = (0,0,255)
green           = (0,255,0)

# DISPLAY SETTING IN PYGAME
gameDisplay = pygame.display.set_mode((display_width, display_height))
pygame.display.set_caption('Car Simulator_Q')
clock = pygame.time.Clock()

# Q-LEARNING
class static:
    FLAG = 1
    REWARD = 0

class Obstacle:
    def __init__(self, x, y, width, height, color):
        self.x          = x
        self.y          = y
        self.width      = width
        self.height     = height
        self.color      = color
```

## Q471.py - Part 2

```
list_obstacles = []
w01 = Obstacle(200,0,600,5,black)   #top wall
list_obstacles.append(w01)
w02 = Obstacle(795,0,5,600,black) #right wall
list_obstacles.append(w02)
w03 = Obstacle(0,595,800,5,black) #bottom wall
list_obstacles.append(w03)
w04 = Obstacle(0,200,5,600,black)   #left wall
list_obstacles.append(w04)

w05 = Obstacle(200,200,400,5,black)   #top wall
list_obstacles.append(w05)
w06 = Obstacle(600,200,5,200,black) #right wall
list_obstacles.append(w06)
w07 = Obstacle(200,400,400,5,black) #bottom wall
list_obstacles.append(w07)
w08 = Obstacle(200,210,5,190,black)   #left wall
list_obstacles.append(w08)

list_target = []
w09=Obstacle(10,10,190,190,red) #target
list_target.append(w09)



#OPEN TXT FILE
f = open("result_Q_4.71_a015.txt", 'w')

#CREATE A WORKBOOK AND ADD A WORKSHEET
workbook = xlsxwriter.Workbook('result_Q_4.71_a015.xlsx')
worksheet = workbook.add_worksheet()
worksheet1 = workbook.add_worksheet()

#QLearning SETTING
class agent:
    count = 0
    n_arr = 0
    rewards = 0
    epsilon = 0.1
    alpha = 0.015
    gamma = 0.8
    iteration = 3000
    count_test = 0
    iter_test = 100
    n_t_arr = 0

Q = []
ACTIONLIST = [1,2,3]

lastState = ()
lastAction = static.FLAG
```

## Q471.py - Part 3

```
carAgent=learning.QLearning(ACTIONLIST, agent.epsilon,
agent.alpha, agent.gamma, agent.iteration)

# REWARD FOR MOVE
def move():
    static.REWARD = -0.1
    update()
    agent.rewards += static.REWARD
    static.REWARD = 0

#SAVING RESULTS TO TEXT FILE
# GAME SIMULATION LOOP
def game_loop():
    pygame.init()

    if(agent.count < agent.iteration):
#        carAgent.epsilon -= carAgent.epsilon/agent.iteration
        agent.count += 1
        print (agent.count)

        data = "iteration = %d \n" % agent.count
        f.write(data)

    else:#TEST count
        carAgent.epsilon = 0#Read only Q table
        agent.count_test +=1

        '''
        if(agent.count_test == 1):
            #Save Q-table####################
            with open('Q-table_204.txt','w') as myfile:
                json.dump(Q,myfile)
            #Save carAgent###################
            with open('carAgent.txt', 'w') as savefile:
                cPickle.dump(carAgent, savefile)
        '''
        print agent.count_test

        data = "iter_test = %d \n" % agent.count_test
        f.write(data)

        if(agent.count_test > agent.iter_test):
            f.close()
            workbook.close()
            pygame.quit()
            sys.exit(0)

    global car, car_group
    car = Sprite.Robot('car.png', (700,500))
    car_group = pygame.sprite.RenderPlain(car)
```

## Q471.py - Part 4

```
gameExit = False

# IN GAME
while not gameExit:
    # USER INPUT
    deltat = clock.tick(fps)
    gameDisplay.fill(white)

    # MEASURE DISTANCE with TARGET, position of car, and speed of car
    global dist, car_position, car_speed
    dist = math.hypot(car.x - (w09.x+50), car.y - (w09.y+50))
    car_position = (car.x, car.y)
    car_speed = car.speed

    #dx = w09.x-car.x
    #dy = w09.y-car.y
    dx = car.x - (w09.x+50)
    dy = car.y - (w09.y+50)
    global rads,degs,rads1,degs1
    rads = (math.atan2(dx,dy))
    degs = math.degrees(rads)
    dist = round(dist/4)
    degs = round(degs)

    # RENDERING
    global list_rect_obstacles, list_rect_target
    list_rect_obstacles = []
    for ob in list_obstacles:
        list_rect_obstacles.append(pygame.draw.rect(gameDisplay,black,(ob.x,ob.y,ob.width,ob.height)))
    list_rect_target = []
    for tar in list_target:
        list_rect_target.append(pygame.draw.rect(gameDisplay,red,(tar.x,tar.y,tar.width,tar.height)))
    car_group.update(deltat)
    car_group.draw(gameDisplay)
    car.draw_rays(gameDisplay)
    gameDisplay.blit(car.image, car.rect)
    pygame.display.flip()
```

## Q471.py - Part 5

```
        for event in pygame.event.get():
            if not hasattr(event, 'key'): continue
            down = event.type == pygame.KEYDOWN
            if event.key == pygame.K_ESCAPE:
                f.close()
                workbook.close()
                sys.exit(0)


        if(static.FLAG == 1):
            car.k_up = -10
            move()
        elif(static.FLAG == 2):
            car.k_up    = -5
            car.k_down  = 0
            car.k_left  = 10
            car.k_right = 0
            move()
        elif(static.FLAG == 3):
            car.k_up    = -5
            car.k_down  = 0
            car.k_left  = 0
            car.k_right = -10
            move()

        #ARRIVAL
        if car.rect.collidelist(list_rect_target) != -1: #if car arrives
            print 'arrive'
            if(agent.count<agent.iteration):
                agent.n_arr += 1
                print "#of success :%d" % agent.n_arr

                data1 = "%d \n" % agent.n_arr
                f.write(data1)


                worksheet.write(agent.count,0,str(agent.count))
                worksheet.write(agent.count,1,str(agent.n_arr))


            static.REWARD = 10
            agent.rewards += static.REWARD
            update()
            static.REWARD = 0
```

## Q471.py - Part 6

```
            #test count
            if(agent.count_test > 0):
                agent.n_t_arr += 1
                print "#of success : %d \n" % agent.n_t_arr
                successrate = (agent.n_t_arr*100/agent.count_test)
                print "success rate : %d \n" % successrate

                data1 = "%d \n" % agent.n_t_arr
                f.write(data1)

                worksheet1.write(agent.count_test,0,str(agent.count_test))
                worksheet1.write(agent.count_test,1,str(agent.n_t_arr))


            game_loop()

        #COLLISION
        if (car.rect.collidelist(list_rect_obstacles) != -1):
            print 'crash'

            worksheet.write(agent.count,0,str(agent.count))
            #worksheet.write(agent.count,1,'collision')

            if(agent.count_test > 0):
                worksheet1.write(agent.count_test,0,str(agent.count_test))
                #worksheet.write(agent.count_test,1,'collision')

            static.REWARD = -10
            update()
            agent.rewards += static.REWARD
            static.REWARD = 0
            game_loop()

# Q LEARNING UPDATE
def update():
    temp = []
    state = temp

    s = car.sense(list_rect_obstacles)
    # STATE DEFINITION
    list_temp = s

    #ULTRASONIC SENSOR DISCRETIZATION
    for n in range(4):
        if(0 <= s[n] < 100):
            list_temp[n] = 0
        elif(100 <= s[n] < 200):
            list_temp[n] = 1
        elif(200 <= s[n] <= 300):
            list_temp[n] = 2
```

## Q471.py - Part 7

```
    #DISTANCE TO TARGET
    if(0 <= dist < 25):
        s_dist = 0
    elif(25 <= dist < 50):
        s_dist = 1
    elif(50 <= dist < 75):
        s_dist = 2
    elif(75 <= dist < 100):
        s_dist = 3
    else:
        s_dist = 4

    #ANGLE TO TARGET
    if(degs<45):
        s_degs = 0
    else:
        s_degs = 1

    list_temp.append(s_dist)
    list_temp.append(s_degs)

    temp = list_temp
        # since distance [] is of no use
    if(agent.count_test > 0):
        state = temp
        state = tuple(state)
        static.FLAG = carAgent.chooseAction(state)
    else:
        if temp != []:
            state = temp
            state = tuple(state)
            action = static.FLAG
            Q.append([state, action])
            lastState = Q[len(Q)-2][0]
            lastAction = Q[len(Q)-2][1]

        carAgent.learn(lastState, lastAction, static.REWARD, state)
        static.FLAG = carAgent.chooseAction(state)

game_loop()
pygame.quit()
sys.exit(0)
```

## From J. Stober [25], tile.py - Part 1

```
#!/usr/bin/python
"""
Author: Jeremy M. Stober
Program: TILES.PY
Date: Monday, March 31 2008
Description: A simple CMAC implementation.
"""

import os, sys, getopt, pdb
from numpy import *
from numpy.random import *
import pylab
from  mpl_toolkits.mplot3d import Axes3D
import pickle

pylab.ioff()

class CMAC(object):
    def __init__(self, nlevels, quantization, beta):
        self.nlevels = nlevels
        self.quantization = quantization
        self.weights = {}
        self.beta = beta

    def save(self,filename):
        pickle.dump(self,open(filename,'wb'),pickle.HIGHEST_PROTOCAL)

    def quantize(self, vector):
        """
        Generate receptive field coordinates for each level of the CMAC.
        """
        quantized = (vector / self.quantization).astype(int)
        coords = []

        for i in range(self.nlevels):
            # Note that the tile size is nlevels * quantization!
            # Coordinates for this tile.
            point = list(quantized - (quantized - i) % self.nlevels)

            # Label the ith tile so that it gets hashed uniquely.
            point.append(i)
            coords.append(tuple(point))
        return coords
```

## From J. Stober [25], tile.py - Part 2

```
    def difference(self, vector, delta, quantized = False):
        """
        Train the CMAC using the difference instead of the response.
        """
        # Coordinates for each level tiling.
        coords = None
        if quantized == False:
            coords = self.quantize(vector)
        else:
            coords = vector

        error = self.beta * delta # delta = response - prediction

        for pt in coords:
            self.weights[pt] += error

        return delta

    def response(self, vector, response, quantized = False):
        """
        Train the CMAC.
        """
        # Coordinates for each level tiling.
        coords = None
        if quantized == False:
            coords = self.quantize(vector)
        else:
            coords = vector

        # Use Python's own hashing for storing feature weights. If you
        # roll your own you'll have to learn about Universal Hashing.
        prediction = sum([self.weights.setdefault(pt, 0.0) for pt in coords]) / len(coords)
        error = self.beta * (response - prediction)

        for pt in coords:
            self.weights[pt] += error

        return prediction

    def __len__(self):
        return len(self.weights)

    def eval(self, vector, quantized = False):
        """
        Eval the CMAC.
        """
        # Coordinates for each level tiling.
        coords = None
        if quantized == False:
            coords = self.quantize(vector)
        else:
            coords = vector

        return sum([self.weights.setdefault(pt, 0.0) for pt in coords]) / len(coords)
```

## Sprite.py - Part 1

```
import pygame, math
import itertools

#from joblib import Parallel, delayed
import multiprocessing

r_visual_range   = 300      #measured from robot center
r_visual_angle   = 60       #in degrees, must divide 90 exactly!
r_visual_granularity = 4    #must be < wall_thickness for walls to be detected correctly!
r_init_azi       = 0
r_step_theta     = 7.5

class Obstacle:      #for now just colored rectangles
    def __init__(self, x, y, width, height, color):
        self.x          = x
        self.y          = y
        self.width      = width
        self.height     = height
        self.color      = color

class Robot(pygame.sprite.Sprite):
    MAX_FORWARD_SPEED = 1
    MAX_REVERSE_SPEED = -5
    ACCELERATION = 2
    TURN_SPEED = 2
    def __init__(self, image, position):
        pygame.sprite.Sprite.__init__(self)
        self.src_image = pygame.image.load(image)
        self.position = position
        self.rect           = self.src_image.get_rect()
        self.rect_original  = self.rect     #unchanging copy, for rotations
        self.x = self.position[0]
        self.y = self.position[1]
        self.speed = 0
        self.direction = 0
        self.k_left = self.k_right = self.k_down = self.k_up = 0
        self.azi            = r_init_azi
        self.visual_range   = r_visual_range
        self.visual_angle   = r_visual_angle
        self.nr_sensors     = 2*90/self.visual_angle+1
        self.retina         = list([self.visual_range]\
                                   for i in range(self.nr_sensors))

    def setAction(self,speed,direction):
        self.speed=speed
        self.direction=direction
        return speed, direction
```

## Sprite.py - Part 2

```
    def update(self, deltat):
        # SIMULATION
        self.speed += (self.k_up + self.k_down)
        if self.speed > self.MAX_FORWARD_SPEED:
            self.speed = self.MAX_FORWARD_SPEED
        if self.speed < self.MAX_REVERSE_SPEED:
            self.speed = self.MAX_REVERSE_SPEED
        self.direction += (self.k_right + self.k_left)
        rad = self.direction * math.pi / 180

        self.azi += (self.k_right + self.k_left)
        '''
        if self.azi >= 360:         #keep theta between -360..360
            self.azi = self.azi-360
        if self.azi <= -360:
            self.azi = self.azi+360
        '''
        self.x += self.speed*math.sin(rad)
        self.y += self.speed*math.cos(rad)
        self.position = (self.x, self.y)
        self.image = pygame.transform.rotate(self.src_image, self.direction)
        self.rect = self.image.get_rect()
        self.rect.center = self.position

    def printRetina(self):
        """Prints the content of the retina list"""
        for s in self.retina:
            if (s[0] == self.visual_range): #this really means >=, since sense() func. caps distances
                                            #to visual_range
                print ('>'+str(self.visual_range))
            else:       #obstacle detected
                print (s)
        print ('\n')

    #this function's job is to place in self.retina the range sensed by each sensor
    def sense(self, list_rect_obstacles):
        n = (self.nr_sensors - 1)/2     #the "natural" sensor range is -n to +n
        granu = r_visual_granularity    #must be at least as large as the wall thickness!!
        for i in range(-n,n+1):         #sense with each of the 2n+1 range sensors
            ang = (self.azi - i*self.visual_angle)*math.pi/180
            for distance in range(granu, self.visual_range+granu, granu):
                x = self.rect.center[0]-distance*math.sin(ang)  #endpoint coordinates
                y = self.rect.center[1]-distance*math.cos(ang)
                nr_collisions = 0
                count = -1          #needed to coordinate the two lists, to extract color after loop
```

## Sprite.py - Part 3

```
                for ob in list_rect_obstacles:  #use the stripped-down list of rectangles for speed
                    count = count + 1
                    if ob.collidepoint(x,y):
                        nr_collisions = 1
                        break       #breaks out of wall loop
                if nr_collisions:   #non-zero collision
                    break           #breaks out of distance loop

            #distance now has the min. between the visual range and the first collision
            #self.retina[i+n][0] = distance
            self.retina[i+n][0] = distance
            '''
            if nr_collisions:       #nr_collisions is 1 if a collision has occurred
                self.retina[i+n][1] = list_obstacles[count].color #color comes form the larger list
            else:
                self.retina[i+n][1] = pygame.Color(color_of_nothing)
            '''
        #print 'sense -->retina is:\n', self.retina
#         self.printRetina()
        #print list_dist
        merged = list(itertools.chain.from_iterable(self.retina))

#         print merged
        return merged

    def draw_rays(self, target_surf):
        n = (self.nr_sensors - 1)/2 #the "natural" sensor range -n to +n
        for i in range(-n,n+1):     #draw the 2n+1 rays of the range sensors
            ang = (self.azi - i*self.visual_angle)*math.pi/180
            x = self.rect.center[0]-self.retina[i+n][0]*math.sin(ang)
            y = self.rect.center[1]-self.retina[i+n][0]*math.cos(ang)
            #use aaline for smoother (but slower) lines
            pygame.draw.line(target_surf, (0,0,0), self.rect.center, (x,y))
```

## I2Cparallaxmotion (Arduino) - Part 1

```
#include <Servo.h>
#include <Wire.h>
#define SLAVE_ADDRESS 0x04

int number=0;
int state=0;
Servo servoLeft;
Servo servoRight;

void setup(){
  Serial.begin(9600);
  // initialize i2c as slave
  Wire.begin(SLAVE_ADDRESS);
  tone(4,3000,1000);
  delay(1000);

  servoLeft.attach(13);
  servoRight.attach(12);

  // define callbacks for i2c communication
  Wire.onReceive(receiveData);
  Wire.onRequest(sendData);
  /*forward(2000);
  turnLeft(600);
  turnRight(600);
  backward(2000);

  disableServos();*/
}

void loop(){
}

void forward(int time){
  servoLeft.writeMicroseconds(1700);
  servoRight.writeMicroseconds(1300);
  delay(time);
}

void turnRight(int time){
  servoLeft.writeMicroseconds(1300);
  servoRight.writeMicroseconds(1300);
  delay(time);
}
```

## I2Cparallaxmotion (Arduino) - Part 2

```
void turnLeft(int time){
  servoLeft.writeMicroseconds(1700);
  servoRight.writeMicroseconds(1700);
  delay(time);
}

void backward(int time){
  servoLeft.writeMicroseconds(1300);
  servoRight.writeMicroseconds(1700);
  delay(time);
}

void disableServos(){
  servoLeft.detach();
  servoRight.detach();
}

// callback for received data
void receiveData(int byteCount){

  while(Wire.available()) {
    number = Wire.read();
    Serial.print("Data received: ");
    Serial.println(number);

    if (number == 1){
      forward(200);
      number=0;
    }else if(number == 2){
      turnLeft(200);
      number=0;
    }else if(number == 3){
      turnRight(200);
      number=0;
    }else if(number == 4){
      backward(600);
      number=0;
    }else if(number == 5){

    }else{
      disableServos();
    }
  }
}

// callback for sending data
void sendData(){
  Wire.write(number);
}
```

# LIST OF FIGURES

# LIST OF TABLES